Structure of a server

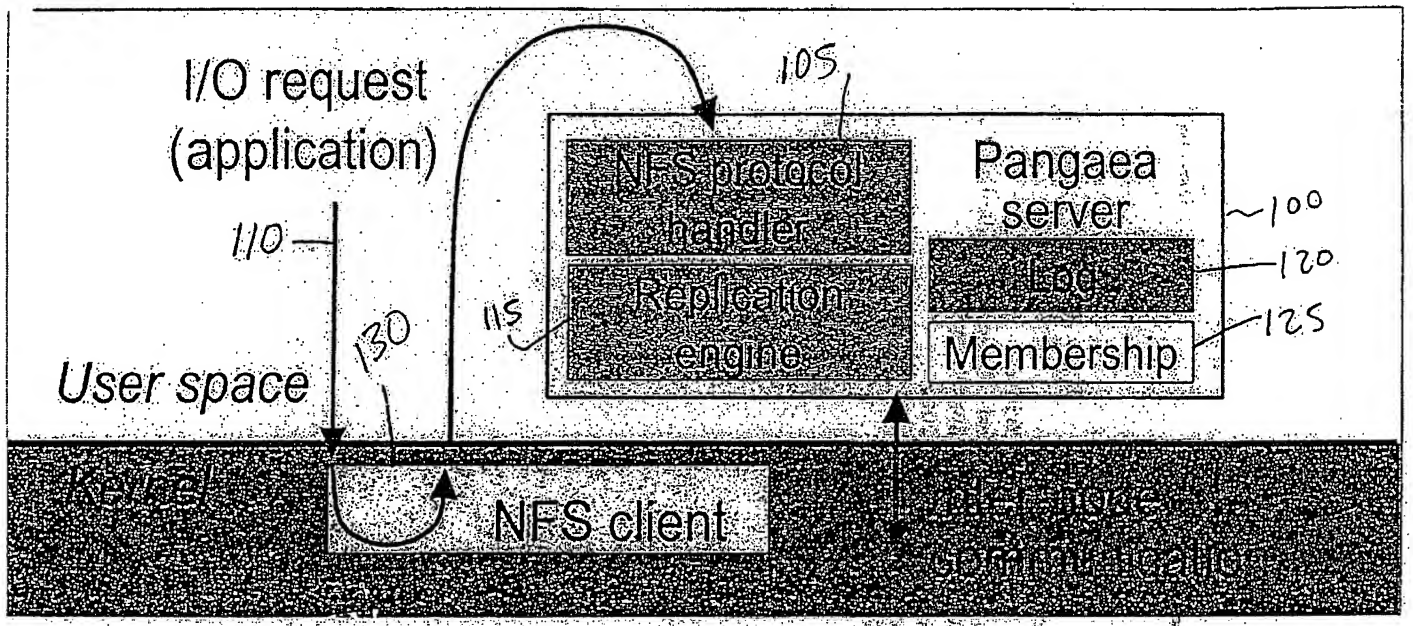


Figure 1

BEST AVAILABLE COPY

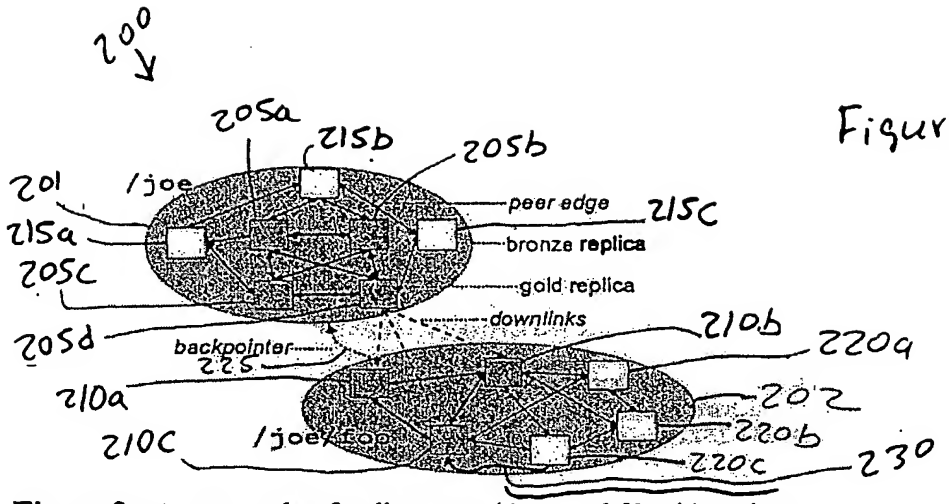


Figure 2

Figure 2: *An example of a directory* `/joe` *and file* `/joe/foo`*. Each replica of* `joe` *stores three pointers to the gold replicas of* `foo`*. Each replica of* `foo` *keeps a backpointer to the parent directory. Bronze replicas are connected randomly to form strongly connected graphs. Bronze replicas also have uni-directional links to the gold replicas of the file, which are not shown here.*

Key attributes of a replica

```
struct Replica
  fid: FileID                        // 96 bit globally unique file ID
  ts: TimeStamp                      // Pair of hphysical clock, IP addri.
  vv: VersionVector                  // Maps IP addr ?! TimeStamp
  goldPeers: Set(NodeID)             // Set of IP addresses
  peers: Set(NodeID)
  backptrs: Set(FileID, String)      // Pair of hdirID, fnamei
. . .
end
struct DirEntry
  fname: String
  fid: FileID
  downlinks: Set(NodeID)
  ts: Timestamp
end
```

Figure 3

proc UpdateReplica
  r_2: **Replica** // New replica contents.
preconditions:
  $\forall \langle pfid, fname \rangle$: $r_2.bptrs \bullet pfid \in \text{dom}(DISK)$
    and $IsLive(r_2) \Rightarrow IsLive(DISK(pfid))$
postconditions:
  $r_2.bptr \neq \{\}$ $^{\langle 3 \rangle}$

  ...

Fig. 4

- "$\langle val_1, \ldots, val_n \rangle$" represents a tuple of values.
- "$\mathbb{P}$ **type**" represents a (possibly empty) set of **type**. "$\mathbb{P}_1$ **type**" represents a nonempty set of **type**. "**Key** $\mapsto$ **Val**" represents a one-to-many mapping from type **Key** to **Val**.
- "dom(F)" returns the domain of function (or mapping) F, and "ran(F)" returns the range of F. For instance,

$$\begin{aligned} \mathrm{dom}(\{1 \mapsto 3, 2 \mapsto 8, 4 \mapsto 3\}) &= \{1,2,4\}, \\ \mathrm{ran}(\{1 \mapsto 3, 2 \mapsto 8, 4 \mapsto 3\}) &= \{3,8\}. \end{aligned}$$

- "X $\oplus$Y" substitutes a part of mapping X by Y. E.g.,

$$\begin{aligned} &\{1 \mapsto 3, 2 \mapsto 1\} \oplus \{1 \mapsto 5, 3 \mapsto 4\} \\ &\quad = \{1 \mapsto 5, 2 \mapsto 1, 3 \mapsto 4\}. \end{aligned}$$

- "X $\lhd$Y" means function-domain restriction. E.g.,

$$\{2\} \lhd \{1 \mapsto 3, 2 \mapsto 8, 4 \mapsto 6\} = \{1 \mapsto 3, 4 \mapsto 6\}.$$

- "$\forall var$: *set* $\bullet$*expr*" means that *expr* holds for *var* in *set*. E.g.,

$$\forall n : \{11, 13, 17\} \bullet \mathrm{IsPrime}(n).$$

- "$\Diamond$ *expr*" means that *expr* holds eventually.
- "{*var*: *set* $\bullet$*expr*}" means set comprehension. E.g.,

$$\{x : \{1,2,3\} \bullet x^2\} = \{1,4,9\}.$$

Fig. 5

```
type Replica = record
    fid⟨1⟩ : FileID
    peers⟨2⟩ : ℙ NodeID
    gpeers⟨3⟩ : ℙ1 NodeID
    bptrs⟨4⟩ : ℙ Backptr
    deadBptr⟨5⟩ : Backptr
    ts⟨6⟩ : Timestamp
type RegularReplica inherits Replica =
    contents⟨7⟩ : Data
    Invariants:
        ¬IsLive(r) ⇒ contents = {}
type DirReplica inherits Replica =
    ents⟨8⟩ : ⟨FileID,String⟩ ⟼ DEntry
    Invariants:
        ¬IsLive(r) ⇒ ents = {}
type Backptr = ⟨FileID, String⟩
type Dentry = record
    valid⟨9⟩ : bool
    ts⟨10⟩ : Timestamp
    gpeers⟨11⟩ : ℙ1 NodeID
proc IsLive(r)
    return r is the root or r.bptrs ≠ {}
```

Fig. 6

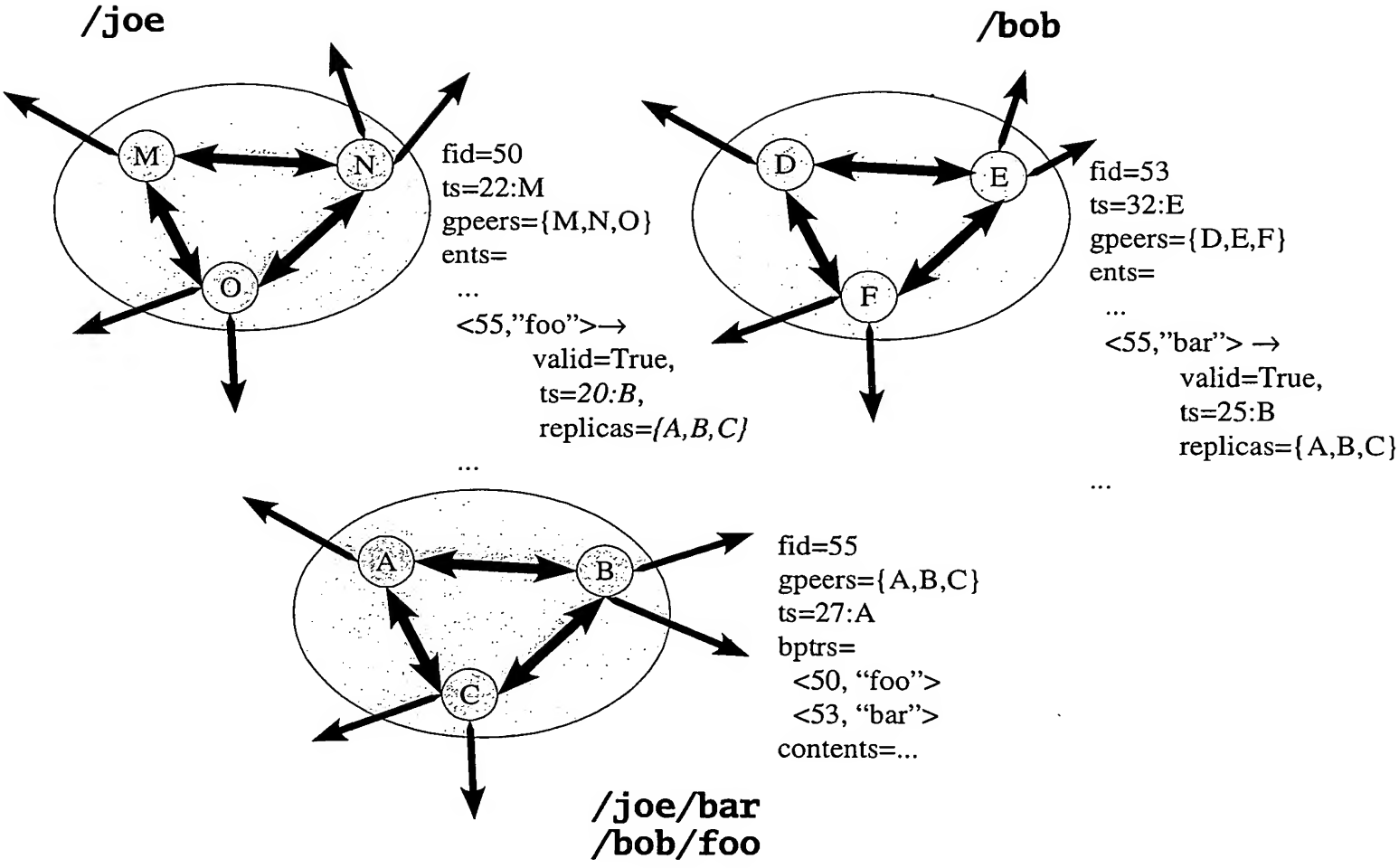

/joe
M
N
O
fid=50
ts=22:M
gpeers={M,N,O}
ents=
...
<55,"foo">→
valid=True,
ts=20:B,
replicas={A,B,C}
...
/bob
D
E
F
fid=53
ts=32:E
gpeers={D,E,F}
ents=
...
<55,"bar"> →
valid=True,
ts=25:B
replicas={A,B,C}
...
A
B
C
fid=55
gpeers={A,B,C}
ts=27:A
bptrs=
<50, "foo">
<53, "bar">
contents=...
/joe/bar
/bob/foo

Fig. 7

DISK: **FileID** $\rightarrowtail$ **Replica**
CLOG: **FileID** $\rightarrowtail \mathbb{P}_1$ **NodeID**
ULOG: **FileID** $\rightarrowtail \mathbb{P}$ **Backptr**
Invariants::
// Updates are only for existing replicas.
$\mathrm{dom}(CLOG) \cup \mathrm{dom}(ULOG) \subseteq \mathrm{dom}(DISK)$ ⟨12⟩

Fig. 8

```
proc Create
    d: DirReplica // The local replica of the parent directory.
    fname: string // The name of the new file in d
    gpeers: ℙ1 NodeID // The placement of the replicas of the file.
preconditions:
    IsLive(d) ⟨13⟩
---------------------------------------------------------------
    r ← Newreplica()
    r.fid ← Newfileid()
    r.gpeers ← gpeers
    r.ts ← Newtimestamp()
    r.peers ← {}
    r.bptrs ← {⟨d.fid, fname⟩}
    r.contents ← None
    UpdateReplica(r)
```

Fig 9

```
proc Unlink
   f: Replica // The file to be unlinked.
   d: DirReplica // The directory the file belongs to.
   fname: string // f's name in d.
preconditions:
   IsLive(d)
   f is a directory ⇒ f.ents = {}
   ⟨d.fid,fname⟩ ∈ f.bptrs
---
   f' ← Deepcopy(f)
   f'.bptrs ← f.bptrs \ {⟨d.fid,fname⟩}
   if f'.bptrs = {} then
      f'.deadBptr ← ⟨d.fid,fname⟩
   f'.ts ← Newtimestamp()
   UpdateReplica(f')
```

Fig. 10

```
proc Hardlink
   f: RegularReplica // The replica of the file.
   d: DirReplica // The directory to which f will be linked to.
   fname: string // The filename within d.
preconditions:
   IsLive(d)
---------------------------------------------------------------
   f' ← Deepcopy(f)
   f'.bptrs ← f.bptrs ∪ {⟨d.fid, fname⟩}
   f'.ts ← Newtimestamp()
   UpdateReplica(f')
```

Fig. 11

```
proc Rename
   f: Replica // The file to be moved.
   d_F: DirReplica // The origin dir.
   d_T: DirReplica // The destination dir.
   fname_F: string // The filename in d_F
   fname_T: string // The filename in d_T
preconditions:
   IsLive(d_F) and IsLive(d_T)
   ⟨d_F.fid, fname_F⟩ ∈ f.bptrs
---------------------------------------------------------------
   f' ← Deepcopy(f)
   f'.bptrs ← f.bptrs \ {⟨d_F.fid,fname_F⟩} ∪ {⟨d_T.fid,fname_T⟩}
   f'.ts ← Newtimestamp()
   UpdateReplica(f')
```

Fig. 12

```
proc Write
    f: RegularReplica
    newcontents: Data
---------------------------------------------
    f' ← Deepcopy(f)
    f'.contents ← newcontents
    f'.ts ← Newtimestamp()
    UpdateReplica(f')
```

Fig. 13

```
proc UpdateReplica
   r2: R plica // New replica contents.
preconditions:
   // All parent directories are stored locally.
   // Moreover, if r2 is live, then parent must also be live.
   ∀ ⟨pfid,fname⟩: r2.bptrs • pfid ∈ dom(DISK)
     and IsLive(r2) ⇒ IsLive(DISK(pfid)) (14)

---

if r2.fid ∉ dom(DISK) then
   // The replica isn't locally stored yet.
   DISK ← DISK ∪ { r2.fid ↦ r2 }
   IssueCupdate(r2)
   return

r1 ← DISK(r2.fid)

if File is regular then
   Do some application-specific stuff.
   We can potentially use version vectors here.
else
   // Union dir entries, taking ones with newer timestamps on conflict.
   for (key ↦ e) ∈ r2.ents
     if key ∉ dom(r1.ents) or r1.ents(key).ts < e.ts
       r1.ents ← r1.ents ⊕ {key ↦ e}
     for each added or deleted entry ⟨fid,fname⟩ in r1.ents
       // Entry ⟨fid,fname⟩ is potentially inconsistent. Fix up later.
       if fid ∈ dom(DISK) then
         IssueUupdate(DISK(fid), {}) (15)

if r2.ts > r1.ts then (16)
   // The file's attributes are to be updated.
   r1.ts ← r2.ts
   if r1.gpeers ≠ r2.gpeers then
     r1.gpeers ← r2.gpeers
     // When the replica's gold-peer set changes, I must reflect the
     // change to the parent dir entry.
     IssueUupdate(r, {})

   // Resolve potential conflicts on back pointers
   if r1.bptrs ≠ r2.bptrs or r1.deadBptr ≠ r2.deadBptr then
     IssueUupdate(r1, r1.bptrs \ r2.bptrs) (17)
     r1.bptrs ← r2.bptrs
     r1.deadBptr ← r2.deadBptr

     // If the last link to the replica is gone, erase the contents.
     if ¬ IsLive(r1) then
       if r1 is a regular file then
         r1.contents ← None
       else
         for e ∈ r1.ents • e.valid
           IssueUupdate(DISK(e.fid), {}) (18)
         r1.ents ← {}

if Any of r1's attributes has changed then
   IssueCupdate(r1) (19)
```

Fig. 14

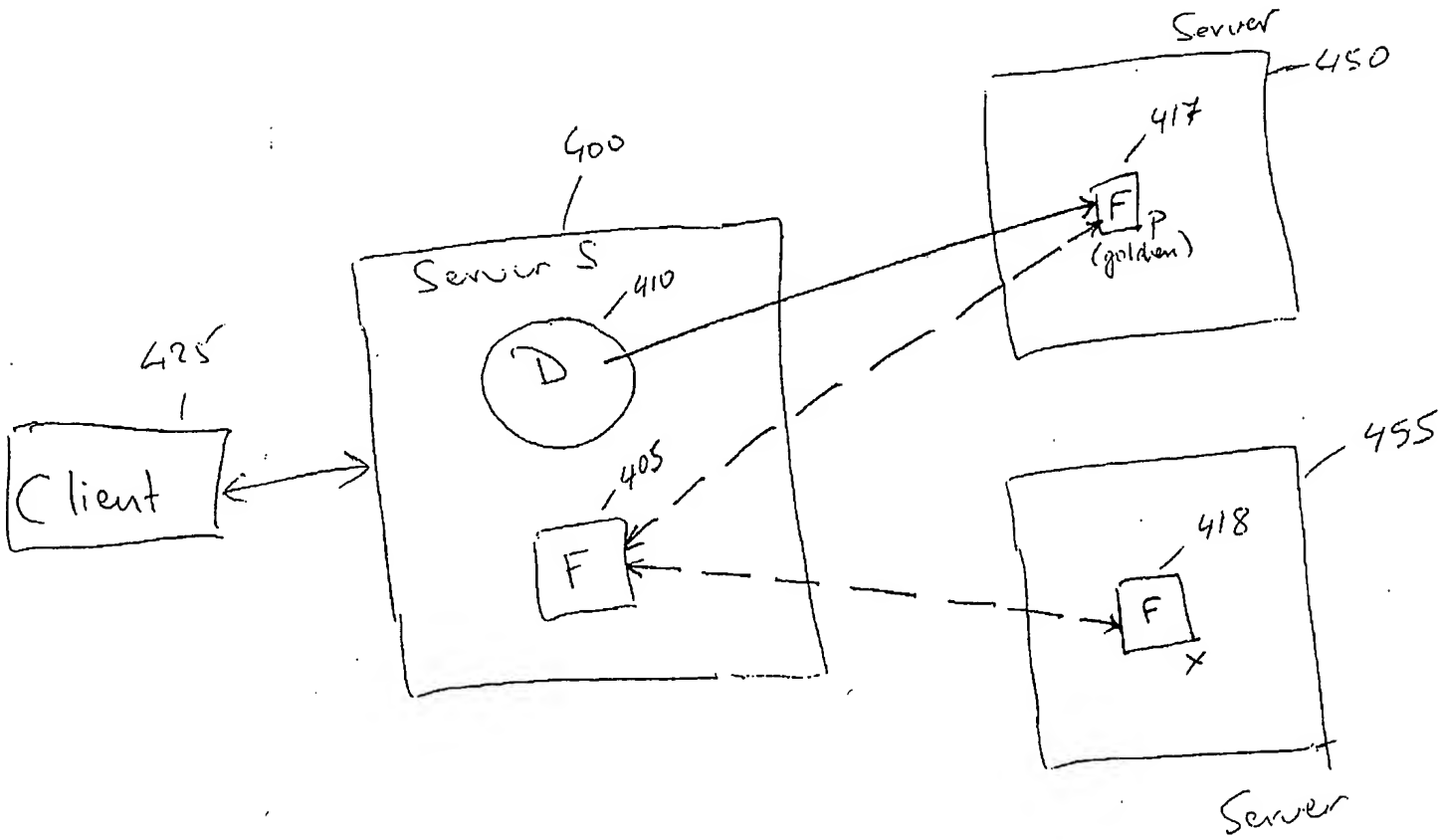

Server
450
417
400
F
P
(golden)
Server S
410
D
425
Client
405
F
455
418
F
X
Server

Figure 15

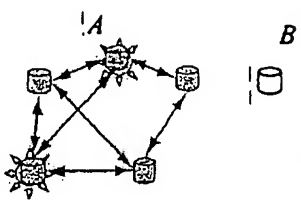
```
Protocol for adding a replica.

Constants:
     M: Number of neighbors per replica.
     MAXHOPS: The number of hops per a random walk (the
usual value is 3)

#
# AddReplica is the main procedure that adds
# a replica of file F on the executing node.
#
AddReplica(F, G)
       G: the set of gold replicas of F.
       (G is obtained by looking up the parent directory)

       g = Pick a random live node in G.
       Send to g, "CreateReplica(F, myself)"
       Wait for the contents to arrive.
       Store contents and reply the client.

       r = find the replica of F.
       Send to g, "StartRandomWalk(F, myself)"
       Wait for the set of neighbors N to arrive.
       for n in N:
           Add edge to n in r.
           Send to n, "AddEdge(F, myself)"

SendReplicaContents(F, Sender):
       F: the ID of the file
       Sender: the node requesting replica creation.

       r = find the replica of F
       n = pick the replica closest to Sender among
graphneighbors of r.
            Send to n, "SendReplicaContents(F, Sender)"

SendReplicaContents(F, Sender):
       F: the ID of the file
       Sender: the node requesting replica creation.

       r = find the replica of F
       Send r to Sender.

StartRandomWalk(F, Sender):
       F: the ID of the file
       Sender: the node requesting replica creation.
```

Figure 16A

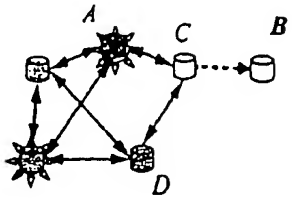
```
        r = find the replica of F
        N = {}
        for i = 0 to M-2:
            n = pick random graph neighbor in r.
         Send to n, "DoRandomWalk(F, 0, myself)"
         Receive nodeid from n.
         Add nodeid to N.
        Send N to Sender.

DoRandomWalk(F, hops, prevHopNode):
        F: the ID of the file
        hops: the number of hops made so far.

        if hops == MAXHOPS
            Send myself to prevHopNode
        else
            r = find the replica of F.
            n = pick random graph neighbor in r
            Send to n, "DoRandomWalk(F, hops + 1, myself)"
            Receive nodeid from n.
            Send nodeid to prevHopNode

AddEdge(F, peer):
        F: the ID of the file
        peer: the node to span edge to
        r = find the replica of F
        Add edge to peer in r
```

Figure 16B

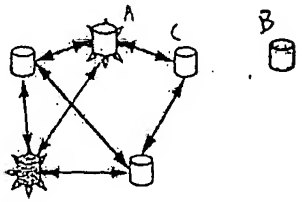
Adding or deleting a bronze replica



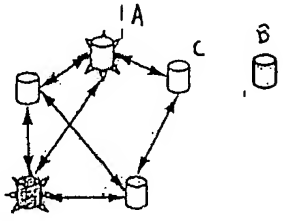
(1) B find gold replica A in the parent dir entry. B asks A to send contents.



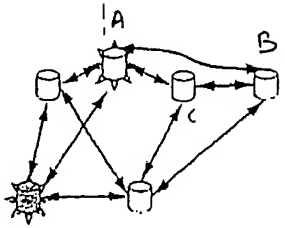
(2) A asks C to send contents to B



(2) Find new peers using random walks starting from A.



(3) Issue RPCs from B to the picked nodes.



(4) Final state.

Figure 17

```
proc IssueCupdate
   r: Replica // The replica of the file being updated.
---
CLOG(r.fid) ← r.gpeers ∪ r.peers
```

```
proc PropagateCupdate // Runs periodically in the background
---
for (fid ↦ targets) ∈ CLOG
   r ← DISK(fid) // See ⟨12⟩.
   // Send the location of the parent dirs so that the target can
   // replicate them to ensure name-space containment.
   pDirs ← {p: r.bptrs • ⟨p.fid, DISK(p.fid).gpeers⟩} // See ⟨14⟩.
   for n ∈ targets
      send ⟨CUPDATE, r, pDirs⟩ to n.
when receive ⟨CUPDATE-REPLY, ts⟩ from node n
   if CLOG(fid).ts = ts
      CLOG(fid) ← CLOG(fid) \ {n}
      Remove fid from CLOG when CLOG(fid) becomes empty
```

```
when receive ⟨CUPDATE, r, pDirs⟩
   r: Replica // New replica contents.
   pDirs: ℙ ⟨FileID, ℙ NodeID⟩ // Name and location of parent dirs.
---
for ⟨pfid, ppeers⟩ ∈ pDirs
   CreateReplica(pfid, ppeers)
   ResurrectDirectory(pfid)
UpdateReplica(r)
send⟨CUPDATE-REPLY, r.ts⟩
```

Fig. 16

```
proc IssueUupdate
   r: Replica // The replica of the file
   del: ℙ ⟨FileID,String⟩ // Backpointers deleted from the replica.
---
if r.fid ∈ dom(ULOG) then
   del ← del ∪ ULOG(r.fid)
ULOG ← ULOG ⊕ {r.fid ↦ del}
```

```
proc ProcessUupdate // Called periodically in the background.
---
for (fid ↦ del) ∈ ULOG
   r ← DISK(fid) // See ⟨12⟩.
   for pfid, fname ∈ del ∪ r.bptrs
      ResurrectDirectory(pfid)
      d ← DISK(pfid)
      valid = pfid ∈ r.bptrs // Is this entry to be added?
      new = (⟨fid,fname⟩ ↦ Dentry(valid, r.ts, r.gpeers)}
      d.ents ← (⟨fid,fname⟩ ⩤ d.ents) ∪ new
      if d.ents has changed
         d.ts ← Newtimestamp()
         IssueCupdate(d)
ULOG ← {}
```

Fig. 19

```
proc CreateReplica
   fid: FileID // The ID of the file
   peers: ℙ1 NodeID // The known set of gold peers of the file
postconditions:
   fid ∈ dom(DISK)
```

```
   if fid ∈ dom(DISK) then
      return
   send ⟨SEND-CONTENTS, fid⟩ to random node n ∈ peers
   Wait until receive ⟨CONTENTS, r, pDirs⟩ from n
   for ⟨pfid, ppeers⟩ ∈ pDirs
      CreateReplica(pfid, ppeers)
   UpdateReplica(r)
   Add edges between r and random existing replicas.
```

```
when receive ⟨SEND-CONTENTS, fid⟩ from node n
```

```
   r ← DISK(fid)
   pDirs ← {p: r.bptrs • ⟨p.fid, DISK(p.fid).gpeers⟩} // See ⟨14⟩.
   send ⟨CONTENTS, r, pDirs⟩ to n.
```

Fig. 20

```
proc ResurrectDirectory
   fid: FileID
preconditions:
   fid ∈ dom(DISK)
   fid is a directory
postconditions:
   IsLive(DISK(fid))
```

```
   r ← DISK(fid)
   if IsLive(r) then
      return

   ResurrectDirectory(r.deadBptr.pfid)
   r.bptrs ← { r.deadBptr }
   r.ts ← Newtimestamp()
   IssueCupdate(r)
   let ⟨pfid, fname⟩ = r.deadBptr •.
      d ← DISK(pfid)
      d.ents(⟨fid,fname⟩) ← Dentry(true, r.ts, r.gpeers) ⟨20⟩
      d.ts ← Newtimestamp()
      IssueCupdate(d)
```

F.5. 21

A	$fid_{/}$ fid_{alice}	5:A 12:A	ents={$\langle fid_{alice}$, "alice", 12:A$\rangle$} ents={*$\langle fid_{bar}$, "bar", 13:B$\rangle$}

Fig. 22

A	$fid_{/}$ fid_{alice} fid_{foo}	5:A 6:A 8:A	ents={⟨fid_{foo}, "foo",...⟩} ents={} bptrs=⟨$fid_{/}$, "foo"⟩
B	$fid_{/}$ fid_{bob} fid_{foo}	5:A 7:B 8:A	ents={⟨fid_{foo}, "foo",...⟩} ents={} bptrs=⟨$fid_{/}$, "foo"⟩

Fig. 23

A	$fid_{/}$ fid_{alice} fid_{foo}	10:A 11:A 12:A	ents={*⟨fid_{foo},"foo",...⟩} ents={⟨fid_{foo}, "foo1", 12:A⟩} bptrs=⟨fid_{alice}, "foo1"⟩
B	$fid_{/}$ fid_{bob} fid_{foo}	5:A 7:B 8:A	ents={⟨fid_{foo}, "foo",...⟩} ents={} bptrs=⟨$fid_{/}$, "foo"⟩

Fig. 24

A	$fid_{/}$ fid_{alice} fid_{foo}	10:A 11:A 12:A	ents={*⟨fid_{foo},"foo",...⟩} ents={⟨fid_{foo}, "foo1", 12:A⟩} bptrs=⟨fid_{alice}, "foo1"⟩
B	$fid_{/}$ fid_{bob} fid_{foo}	9:B 11:B 12:B	ents={*⟨fid_{foo}, "foo",...⟩} ents={⟨fid_{foo}, "foo2", 12:B⟩} bptrs=⟨fid_{bob}, "foo2"⟩

Fig. 25

A	$fid_{/}$	10:A	ents={*⟨fid_{foo},"foo",...⟩}
	fid_{alice}	14:A	ents={*⟨fid_{foo}, "foo1", 12:B⟩}
	fid_{bob}	11:B	ents={⟨fid_{foo}, "foo2", 12:B⟩}
	fid_{foo}	12:B	bptrs=⟨fid_{bob}, "foo2"⟩
B	$fid_{/}$	9:B	ents={*⟨fid_{foo}, "foo",...⟩}
	fid_{bob}	11:B	ents={⟨fid_{foo}, "foo2", 12:B⟩}
	fid_{foo}	12:B	bptrs=⟨fid_{bob}, "foo2"⟩

Fig. 26

A	$fid_{/}$	10:A	ents={*⟨fid_{foo},"foo",...⟩}
	fid_{alice}	14:A	ents={*⟨fid_{foo}, "foo1", 12:B⟩}
	fid_{bob}	11:B	ents={⟨fid_{foo}, "foo2", 12:B⟩}
	fid_{foo}	12:B	bptrs=⟨fid_{bob}, "foo2"⟩
B	$fid_{/}$	10:A	ents={*⟨fid_{foo}, "foo",...⟩}
	fid_{bob}	11:B	ents={⟨fid_{foo}, "foo2", 12:B⟩}
	fid_{foo}	12:B	bptrs=⟨fid_{bob}, "foo2"⟩

Fig. 27

A	$fid_{/}$ fid_{foo}	5:A 6:A	ents={⟨fid_{foo}, "foo", 6:A⟩} bptrs={⟨fid_{foo}, "foo"⟩}
B	$fid_{/}$ fid_{foo}	5:A 6:A	ents={⟨fid_{foo}, "foo", 6:A⟩} bptrs={⟨fid_{foo}, "foo"⟩}

Fig. 28

A	$fid_{/}$ fid_{foo}	10:A 11:A	ents={*⟨fid_{foo}, "foo", 11:A⟩} bptrs={}
B	$fid_{/}$ fid_{foo}	5:A 6:A	ents={⟨fid_{foo}, "foo", 6:A⟩} bptrs={⟨fid_{foo}, "foo"⟩}

Fig. 29

A	$fid_{/}$ fid_{foo}	10:A 11:A	ents={*⟨fid_{foo}, "foo", 11:A⟩} bptrs={}
B	$fid_{/}$ fid_{foo}	5:A 11:B	ents={⟨fid_{foo}, "foo", 6:A⟩} bptrs={⟨fid_{foo}, "foo"⟩}

Fig. 30

A	$fid_{/}$ fid_{foo}	12:A 11:B	ents={⟨fid_{foo}, "foo", 11:B⟩} bptrs={⟨fid_{foo}, "foo"⟩}
B	$fid_{/}$ fid_{foo}	12:A 11:B	ents={⟨fid_{foo}, "foo", 11:B⟩} bptrs={⟨fid_{foo}, "foo"⟩}

Fig. 31

A	$fid_{/}$ fid_{foo}	12:B 11:A	ents={*⟨fid_{foo}, "foo", 11:A⟩} bptrs={}
B	$fid_{/}$ fid_{foo}	12:B 11:A	ents={*⟨fid_{foo}, "foo", 11:A⟩} bptrs={}

Fig. 32

A	$fid_{/}$ fid_{foo}	5:A 6:A	ents={⟨fid_{foo}, "foo", 6:A⟩} ents={}, bptrs={⟨$fid_{/}$,"foo"⟩}
B	$fid_{/}$ fid_{foo}	5:A 6:A	ents={⟨fid_{foo}, "foo", 6:A⟩} ents={}, bptrs={⟨$fid_{/}$,"foo"⟩}

Fig. 33

A	$fid_{/}$	5:A	ents={⟨fid_{foo}, "foo", 6:A⟩}
	fid_{foo}	10:A	ents={⟨fid_{bar}, "bar", 11:A⟩}, bptrs={⟨$fid_{/}$,"foo"⟩}
	fid_{bar}	11:A	bptrs={⟨fid_{foo},"bar"⟩}
B	$fid_{/}$	5:A	ents={⟨fid_{foo}, "foo", 6:A⟩}
	fid_{foo}	6:A	ents={}, bptrs={⟨$fid_{/}$,"foo"⟩}

Fig. 34

A	$fid_{/}$	5:A	ents={⟨fid_{foo}, "foo", 6:A⟩}
	fid_{foo}	10:A	ents={⟨fid_{bar}, "bar", 11:A⟩}, bptrs={⟨$fid_{/}$,"foo"⟩}
	fid_{bar}	11:A	bptrs={⟨fid_{foo},"bar"⟩}
B	$fid_{/}$	8:B	ents={*⟨fid_{foo}, "foo", 10:B⟩}
	fid_{foo}	10:B	ents={}, bptrs={}

Fig. 35

A	$fid_{/}$	8:B	ents={*⟨fid_{dir}, "dir", 10:B⟩}
	fid_{dir}	10:B	ents={}, bptrs={}
	fid_{foo}	11:A	bptrs={⟨fid_{dir},"foo"⟩}
B	$fid_{/}$	8:B	ents={*⟨fid_{dir}, "dir", 10:B⟩}
	fid_{dir}	10:B	ents={}, bptrs={}

Fig. 36

A	$fid_{/}$	12:A	ents={⟨fid_{dir}, "dir", 13:A⟩}
	fid_{dir}	13:A	ents={⟨fid_{foo}, "foo", 11:A⟩}
	fid_{foo}	11:A	bptrs={⟨fid_{dir}, "foo"⟩}
B	$fid_{/}$	12:A	ents={⟨fid_{dir}, "dir", 13:A⟩}
	fid_{dir}	13:A	ents={⟨fid_{foo}, "foo", 11:A⟩}

Fig. 37

A	$fid_{/}$	8:B	ents={*⟨fid_{dir}, "dir", 10:B⟩}
	fid_{dir}	10:A	ents={⟨fid_{foo}, "foo", 11:A⟩}, bptrs={⟨$fid_{/}$,"dir"⟩}
	fid_{foo}	11:A	bptrs={⟨fid_{dir},"foo"⟩}
B	$fid_{/}$	8:B	ents={*⟨fid_{dir}, "dir", 10:B⟩}
	fid_{dir}	10:B	ents={}, bptrs={}

Fig. 3B

A	$fid_{/}$	8:B	ents={*⟨fid_{dir}, "dir", 10:B⟩}
	fid_{dir}	10:A	ents={⟨fid_{foo}, "foo", 11:A⟩}, bptrs={⟨$fid_{/}$,"dir"⟩}
	fid_{foo}	11:A	bptrs={⟨fid_{dir},"foo"⟩}
B	$fid_{/}$	8:B	ents={*⟨fid_{dir}, "dir", 10:B⟩}
	fid_{dir}	10:A	ents={⟨fid_{foo}, "foo", 11:A⟩}, bptrs={⟨$fid_{/}$,"dir"⟩}

Fig. 39

A	$fid_{/}$	13:A	ents={⟨fid_{dir}, "dir", 10:A⟩}
	fid_{dir}	10:A	ents={⟨fid_{foo}, "foo", 11:A⟩}, bptrs={⟨$fid_{/}$,"dir"⟩}
	fid_{foo}	11:A	bptrs={⟨fid_{dir},"foo"⟩}
B	$fid_{/}$	13:A	ents={⟨fid_{dir}, "dir", 10:A⟩}
	fid_{dir}	10:A	ents={⟨fid_{foo}, "foo", 11:A⟩}, bptrs={⟨$fid_{/}$,"dir"⟩}

Fig. 40

```
// Called every third night for every replica on the node.
proc GarbageCollection
   LiveNodes: ℙ1 NodeID // Set of live nodes.
   r: Replica // Replica to be inspected.
   EXPIRE: integer // Dead-replica expiration period, e.g., a month.
---
// Remove old tombstones. Removing after EXPIRE seconds is safe
// because we cannot receive any new update with timestamp older
// than EXPIRE after removing r.
if ¬ IsLive(r) and r.ts < Newtimestamp - EXPIRE then
   DISK ← {r.fid} ⩤ DISK
   return

r' ← Deepcopy(r)

// Remove dead entries in the directory
if r' is a directory then
   for (key ↦ val): r'.ents •
      if ¬ val.valid and val.ts < Newtimestamp() - EXPIRE then
         r'.ents ← {key} ⩤ r'.ents
   if at least one entry has been removed from r'.ents then
      r'.ts ← NewTimestamp()
      UpdateReplica(r)

// If some gold peers are found dead, recreate one elsewhere
if me ∈ r.gpeers and r.gpeers ⊈ Livenodes then
   newNodes ← Pick ||r.gpeers \ Livenodes|| random live nodes.
   r'.gpeers ← r.gpeers \ Livenodes ∪ newNodes
   r'.ts ← NewTimestamp()
   UpdateReplica(r)

// If we find graph edges to dead nodes, re-span it.
for n ∈ r.peers \ Livenodes
   Add edges between r and a random replica.
   r.peers ← r.peers ∩ Livenodes
```

Fig. 41

$\forall$ *f:* ran(*DISK*) **and** IsLive(*f*) •
($\langle$*d.fid, fname*$\rangle \in$ *f.bptrs*
$\Rightarrow$ $d \in$ dom(*DISK*) **and** IsLive(*d*))

Fig. 42

1

$\forall$ *d:* ran(*DISK*) **and** IsLive(*d*) •
(*f:* ran(*DISK*) **and** IsLive(*f*) •
(⟨*d.fid,fname*⟩$\mapsto$ *ent*) $\in$ *d.ents* **and** *ent.valid*
and ⟨*d.fid,fname*⟩ $\notin$ *f.bptr*
$\Rightarrow$ $\Diamond$ ⟨*f.fid,fname*⟩$\mapsto$ *ent*) $\notin$ *d.ents*)

Fig. 43

$\forall$ *d:* ran(*DISK*) **and** IsLive(*d*) •
($\langle$*f.fid,fname*$\rangle \mapsto$ *ent*) $\in$ *d.ents* **and** *ent.valid*
$\Rightarrow$ *f:* ran(*DISK'*) **and** IsLive(*f*))

Fig. 44

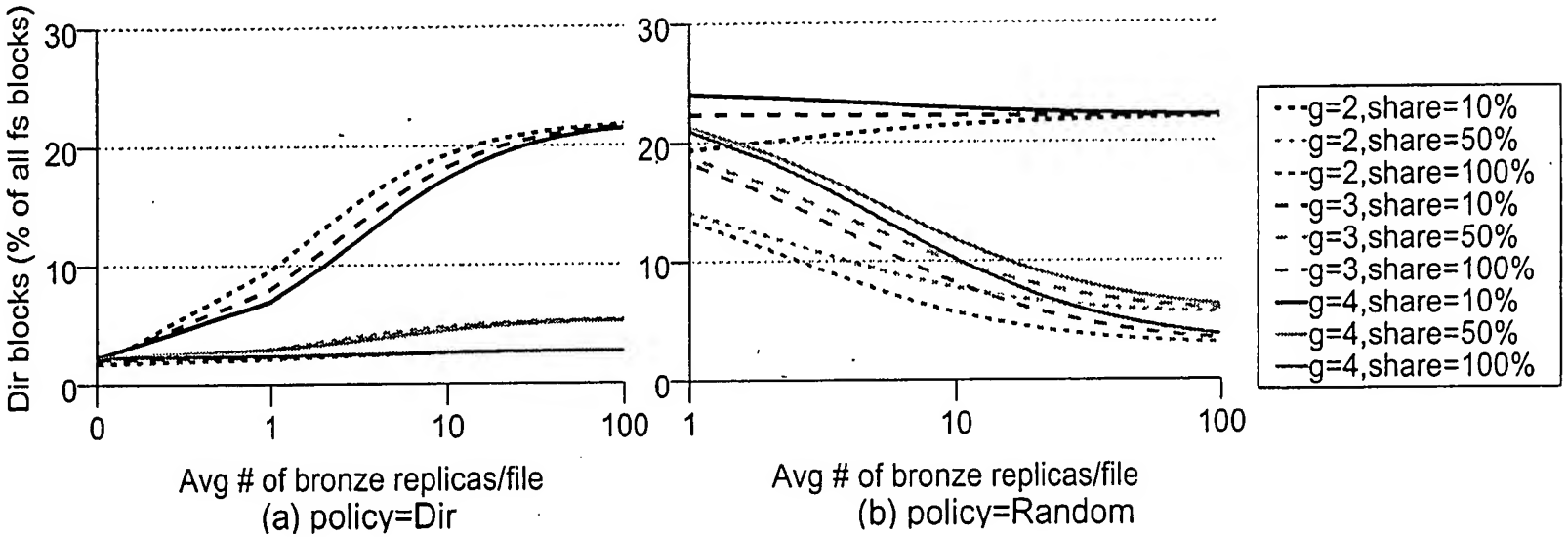

Dir blocks (% of all fs blocks)
30
20
10
0
0
1
10
100
Avg # of bronze replicas/file
(a) policy=Dir
30
20
10
0
1
10
100
Avg # of bronze replicas/file
(b) policy=Random
g=2,share=10%
g=2,share=50%
g=2,share=100%
g=3,share=10%
g=3,share=50%
g=3,share=100%
g=4,share=10%
g=4,share=50%
g=4,share=100%

Fig. 45